Big Data Scenarios Simulator for Deep Learning Algorithm Evaluation for Autonomous Vehicle

Jun Wei Dickson Ang*, Chee Kiat Seow†, Karthikk Subramanian‡ and Sugiri Pranata§
University of Glasgow, UK
Email: *2427283A@student.gla.ac.uk, †CheeKiat.Seow@glasgow.ac.uk
Panasonic R&D Center Singapore (PRDCSG), Singapore
Email: ‡karthikk.subramanian@sg.panasonic.com, §sugiri.pranata@sg.panasonic.com

***Abstract*—One of the challenges in developing autonomous vehicles (AV) is the collection of suitable real environment data for the training and evaluation of machine learning algorithms for autonomous vehicles. Such environment data collection via various sensors mounted on AV is big data in nature which require massive time and money investment and in some specific scenarios could pose a significant danger to human lives. This necessitates the virtual scenarios simulator to simulate the real environment by generating big data images from a virtual fisheye lens that can mimic the field of view and radial distortion of commercial available camera lens of any manufacturer and model. In this paper, we proposed the novelty of developing a fisheye lens with distortion system to generate big data scenarios images to train and test imaged based sensing functions and to evaluate scenarios according to EuroNCAP standards. A total of 10,123 RGB, depth and segmentation images of varying road scenarios were generated by proposed system in approximately 14 hours as compared to existing methods of 20 hours, achieving 42.86% improvement.**

I. Introduction

Big data is increasingly involved in the development of self driving or autonomous vehicles (AV) [1] especially with the advent of 5G technology and applications [2]–[4]. As manual collection of high volume of data required for training, testing and evaluating a machine learning algorithm [5], [6] used in AV is highly unrealistic with so many possible environment scenarios, one viable way to address is the use of computer graphic simulator systems such as CARLA [7], CAIAS [8] and AirSim [9]. These systems aim to simulate a virtual environment in which a virtual car is fitted with virtual sensors to simulate real-world road scenarios. These sensors try to mimic their real-world counterparts in the collection of environmental big data such as RGB, depth and LIDAR information. Other ways exist, such as Richter et al. [10] using the computer game Grand Theft Auto V, and Gaidon et al. [11] using virtual worlds created from a few seed real-world video sequences to generate annotated image datasets.

Existing work on fisheye lens rendering in traditional real-time graphics like Unreal Engine 4, the game engine that CARLA and AirSim is built on, relies on traditional real-time graphics for its rendering system such as Ott et al. [12] stitches separate images captured from 5 different camera angles using a external image processing software called ”glom” to achieve a fisheye image. Bourke et al. [13], on the other hand, generates fisheye images directly within a game engine that applies images captured from 4 different angles onto a static mesh designed to result in a fisheye projection when seen directly. This eliminates external image processing software. The requirement for a fisheye lens on the sensors stems from Euro NCAP standard 2020, which stipulates that objects at a perpendicular distance of 16.7m away from the AV must be detected. At this distance and beyond, objects are outside the standard 90-degree field of view of sensors. A fisheye lens, with its 180-degree field of view, may allow objects to be detected. None of the modern simulators currently available have fisheye lens rendering with accurate distortion on their virtual sensors. In this paper, we propose a fisheye lens with distortion model that addresses the above issues and evaluate in terms of execution time and object detection accuracy with a deep learning model based on a Fully Convolutional One Stage (FCOS) object detection model [6] that achieved consistently high AP and mAP scores of 99.4% and 90.1% respectively. Section II outlines the system overview and implementation of our proposed solution. Section III evaluates the performance of the generated dataset with the deep learning model, followed by our conclusion in Section IV.

II. System Overview & Implementation

CARLA and Unreal Engine 4 were used for the implementation of our proposed system with CARLA implemented as a client-server architecture as shown in Figure 1. The client consists of Python scripts that control weather conditions, camera and sensor setup as well as the logic of the actors in the environment such as pedestrians and vehicles. The server is implemented in Unreal Engine 4 running the actual simulation logic and rendering. Configurations and commands are sent from the client Python scripts to the simulation server for processing. CARLA exposes Python APIs that are

Fig. 1: Architecture of CARLA

wrappers around CARLA's C++ library which handles the communication with the simulation server through TCP. The sensors are configured and set up in client Python scripts. For instance, camera sensors can be attached to a spawned car actor at fixed locations to mimic a real-world setup. These sensors come with a listen method that allows callbacks to be attached. The callback method is invoked whenever new data from the CARLA simulation server is made available to the client. The server listens for commands and configurations from client Python scripts and processes them accordingly. The server also sends sensor information gathered from the simulation to the client Python scripts. In each simulation update, the server will render each camera-based sensor according to client Python scripts' configuration. The rendered image data is sent back to the client to choose how it wants to process this data. Unreal Engine 4 comes with a modern real-time graphics engine which supports physically based rendering techniques and has a robust material system that accurately represents real-world materials. The custom fisheye lens sensor was implemented in Unreal Engine 4 and added to CARLA's sensor registry to allow client Python scripts to configure and attach the sensor like any of the built-in sensor types. For consistency and accuracy, the implemented custom fisheye lens allows configuration of both field of view (FOV), as well as lens distortion. This ensures it can mimic the image output of a real-world camera system fitted with a fisheye lens of any chosen manufacturer and model. As shown in Figure 2, Unreal Engine's graphics pipeline was used to render the environment and create the desired fisheye image. A feature of Unreal Engine 4, the SceneCaptureCube component is used to capture a 360-degree view of the scene. The captured scene information will then be sampled with a pixel shader to create the desired fisheye image.

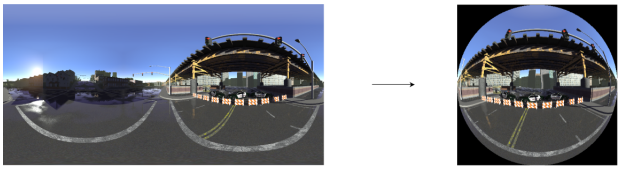


Fig. 2: Custom fisheye lens rendering

In this paper, an RGB, depth and semantic segmentation fisheye lens sensor was used. The fisheye lens sensors are attached to a virtual car that is placed in the environment at the required locations. The placement of the actors in the environment is randomised according to the Euro NCAP 2020 standard. The generated images are saved to local storage for use later. In every five seconds, previously added actors are removed and a new set of stationary vehicles, pedestrians or cyclists are setup randomly in the world, according to the Euro NCAP 2020 requirements. The five-second delay is added between each road scenario to allow time for the actors to stabilise in their positions. A snapshot of the sensor output is then taken and saved to storage in .PNG format. The whole process repeats till the desired number of images have been generated.

As neither Unreal Engine 4 nor CARLA natively supports a fisheye lens rendering scheme, we propose a common technique used in computer graphics to capture multiple camera views known as cube map rendering. Cube map rendering will render the environment at a given camera position with six images, one for each side of a cube. Each side of this cube map is rendered by a camera with a 90-degree field of view and 1:1 aspect ratio. Together, the six images form a complete 360-degree capture of the environment at a given point in space. The first step in creating a fisheye image that mimics the output of a real-world camera system is to generate a cube map of our environment from a specified point or camera position. Unlike traditional cube map rendering, our cube map is rendered to a texture in a latitude-longitude format that not only maximises the usage of the target texture, making use of the entire surface to store image data as shown in the left image in Figure 2 but also allows an easy and direct way to lookup any given point on the cube map by using the latitude-longitude coordinate system. The longitude runs horizontally across the u texture coordinate axis representing a range of 0 to 2π while the latitude runs vertically along the v texture coordinate axis to represent the range of 0 to π.

Given the spherical coordinates of a location, it is trivial to sample the corresponding pixel value from the cube map texture in Lat-Long format. A 3D Cartesian unit-directional vector can also be converted to spherical coordinates and vice versa. The use of the Lat-Long Format also reduces the likelihood of seams during incorrect sampling. Unreal Engine 4 natively supports capturing a cube map using the Lat-Long format using a component called the SceneCaptureCube. The rendered cube map is stored in a cube render target that allows easy sampling given a 3D Cartesian unit direction vector. For this paper, a texture resolution of 1024x1024 was chosen as the render target resolution to offer the best quality in the final output image. The resolution is chosen to provide the best trade-off in terms of rendering speed and image quality.

For each pixel in the final fisheye image, we can compute the corresponding pixel in the captured cube map to sample from. This is achieved by first constructing a 3D Cartesian unit direction vector from the camera position into our environment for every point in the final image. In Unreal Engine 4, the *uv* texture coordinates start at the top left of the image plane at $(0, 0)$ with the *u-axis* going right and *v-axis* going down. The *uv* texture coordinates have a range from 0 to 1. As the final fisheye image should be centred and extends to the edge of the target image plane forming a unit circle, we need to remap Unreal Engine 4's default texture coordinates to one that we desire. The desired coordinate system starts in the middle of the image plane as $(0, 0)$ with *x-axis* going right and *y-axis* going up. Both axes have a range of -1 to 1. This can be achieved with the following:

$$x = 2u - 1, \quad y = -2v + 1 \tag{1}$$

Next, we can convert the Cartesian coordinates (x, y) to 2D

polar coordinates (r, θ).

$$r = \sqrt{x^2 + y^2}, \quad \theta = \tan^{-1}(y/x) \tag{2}$$

Unreal Engine 4 uses a left handed coordinate system with x-axis pointing forwards, y-axis to the right and z-axis upwards. Using θ from (2) and f, the FOV of the fisheye lens, we can then calculate the 3D cartesian direction vector into our environment with (3), (4).

$$\phi = rf/2, \text{ where } f = \text{field of view of lens} \tag{3}$$

$$x = \cos(\phi), \quad y = \cos(\theta)\sin(\phi), \quad z = \sin(\theta)\sin(\phi) \tag{4}$$

The cube map render target generated by the SceneCaptureCube component can then be sampled to retrieve the corresponding environment pixel value. In Unreal Engine 4, sampling of a cube map is done with a 3D Cartesian direction vector. Any pixel that lies outside of the unit circle will be rendered black to denote that it lies outside of the fisheye lens. This creates a perfect fisheye lens image. However, in the real world, due to the difficulties of creating a perfect lens, the lens does not follow a precise linear relationship between the radius on the image plane and the latitude, ϕ [14]. As shown in Figure 3, this can be seen as a slight compression around the periphery of the lens as compared to a perfect lens of a similar FOV.

(a) (b)

Fig. 3: Perfect fisheye image (left) and real fisheye image(right) [14].

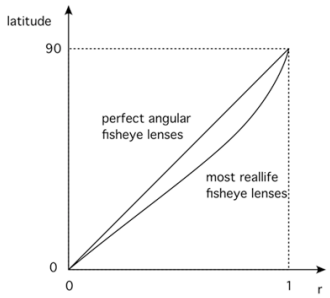


Fig. 4: Relationship between a perfect fisheye lens and the real world [15].

We can approximate the non-linear relationship as shown in Figure 4 between the radius and latitude, ϕ of real fisheye lens, by replacing the linear equation (3) by the 4th order polynomial equation (5).

$$\phi = ar + br^2 + cr^3 + dr^4 \tag{5}$$

Sometimes, lens manufacturers will make available the coefficients of this 4th order polynomial, which allows photographers to correct for the non-linear relationship. In other cases, where this data is not available, careful physical measurement of the optical properties will be required to find the coefficients. In this paper, a Sigma f3.5, 8mm, 180-degrees fisheye lens was used as the reference, and it has the following lens profile as show in Figure 5.

$$\phi = 0.7856r + 3.4708r^2 - 5.8758r^3 + 3.178r^4 \tag{6}$$

In order to perform the sampling of the cube map texture

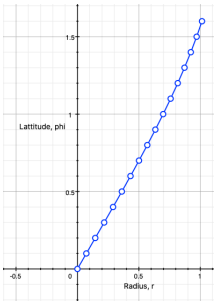


Fig. 5: Lens profile of Sigma f3.5, 8mm, 180-degrees fisheye lens.

into a fisheye image according to the formulas mentioned above, Unreal Engine 4's material system was used. The material system provides a visual scripting tool that allows us to create a custom pixel shader as shown in Figure 6 and Figure 7. This shader will then be used at runtime and is run on each target pixel of our final fisheye image to generate the desired results. The final image is an RGB fisheye image with lens distortion that mimics the real-world output of a camera system fitted with a Sigma f3.5, 8mm, 180-degrees lens as shown in Figure 8. Unreal Engine 4's SceneCaptureCube component allows us to capture not just RGB but depth and stencil values as a cube map texture. Using the same technique as RGB, depth and stencil cube map textures will produce a depth and semantic segmentation fisheye image, respectively.

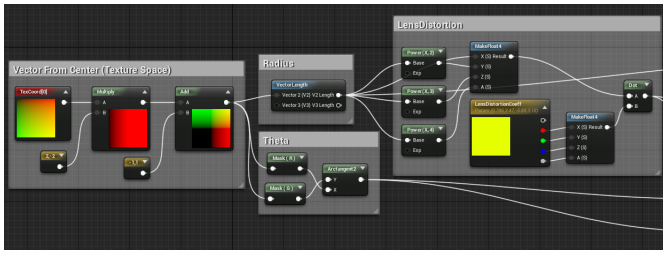


Fig. 6: First half of the material used to render a fisheye image.

The generation of the required dataset is done through a client Python script with CARLA Python APIs. CARLA

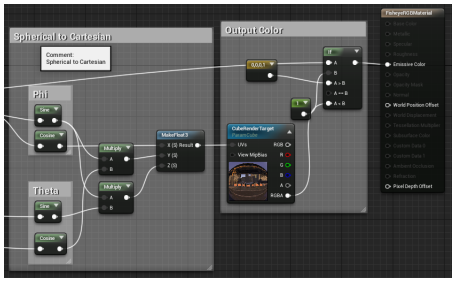


Fig. 7: Second half of the material used to render a fisheye image.

Fig. 8: The final fisheye lens rendering with distortion.

Python APIs allow for the manipulation of various aspects of the simulation. The starting position of a vehicle, sensors to use, number of vehicles and pedestrians in the world, can be configured accordingly. A single script, generate_images.py was created that can generate either the training or testing datasets depending on the script arguments which are set. The generation script has a few key components.

There are several scenarios listed as part of the Euro NCAP 2020 standard. However, in this paper, we are only interested in the Vulnerable Road User (VRU) scenario as shown in Figure 9. This scenario is chosen due to its criticality in blind

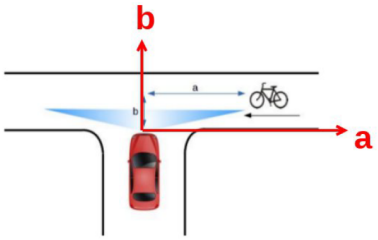


Fig. 9: Euro NCAP scenario used on this project

spot evaluation consideration. Two parameters a, b control the horizontal and vertical distances a pedestrian or cyclist is from a vehicle respectively. This particular scenario was chosen as the object to be detected, either a pedestrian or cyclist is placed on the far side of a vehicle's FOV. The fisheye lens sensor designed with wider FOV is well-suited for capturing objects at this scenario. The CARLA provided world, Town03 is chosen as the world used for dataset generation. As shown in Figure 10, Town03 contains quite a number of T-junctions and intersections, which is particular suited for testing the Euro NCAP scenario shown in Figure 9. The simulation is

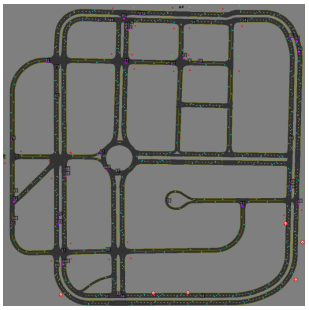
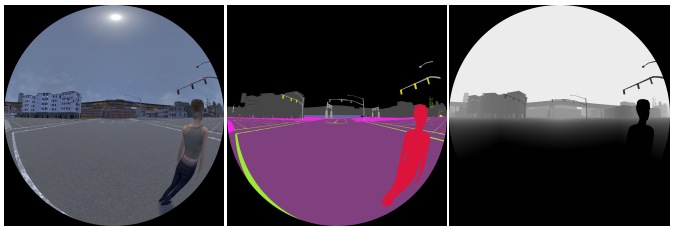


Fig. 10: Layout of Town03.

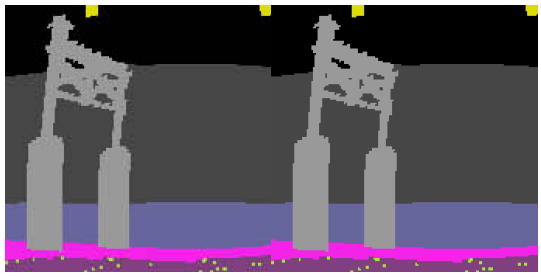
configured to run in synchronous mode with a fixed time step of 15 frames per second. The use of synchronous mode ensures that the simulation server and client Python script is in lockstep with each other to avoid a potential desynchronization. The weather is set according to the script arguments supplied during the execution of the script, with the default weather of clear noon. For consistency, the same vehicle to attach sensors onto is used. Before the script generates each image, the world is configured to match the Euro NCAP 2020 scenario. The primary vehicle with sensors attached is teleported to the desired position. When generating a training dataset, a random location in Town03 is chosen as the position of the primary vehicle. While generating a test dataset, a fixed T junction in Town03 is used instead. After which, either a pedestrian or cyclist is spawned near the primary vehicle depending on the script arguments passed during execution. The pedestrian or cyclist is spawned according to the Euro NCAP 2020 standard seen in Figure 9. Parameters for a and b distances are taken from the script arguments, which are set before script execution. In each update, the world class will compute the time since the last snapshot of RGB, depth, and semantic segmentation images were taken. If five seconds have elapsed, a snapshot is taken. Once the required images are saved, a new scenario is generated as described previously. Three fisheye sensors, RGB, depth and semantic segmentation, are configured for attachment onto the primary vehicle. The sensors are attached to the front of the primary



(a) (b) (c)

Fig. 11: A single road scenario generated by the system. Showing RGB, semantic segmentation and depth images.

vehicle. Every five seconds, the latest set of RGB, depth and semantic segmentation fisheye images received from the simulation server are saved. The images are saved using the .PNG format and are labelled according to their type as well as an incremental counter. As shown in Figure 12, the .PNG image format was chosen as it offers lossless compression as compared to other formats such as .JPG. This is particularly important for semantic segmentation images as they need to be pixel accurate. The compression artefacts introduced by .JPG compression is undesirable.



(a) (b)

Fig. 12: Section of semantic segmentation image enlarged by 300% to show .JPG compression (left) and .PNG lossless compression (right).

III. Evaluation & Results

The generation speed of the dataset using our system was found to be about 40% faster than Richter et al. [10] , which generated 24,996 images with pixel-level semantic segmentation in about 49 hours. Our proposed system generates the same amount of images with the same pixel-level semantic segmentation in about 35 hours. It takes approximately 14 hours to generate 10,123 RGB, depth and segmentation images of varying road scenarios as compared to existing methods of 20 hours, achieving 42.86% improvement. Out of these 10,123 images, 8017 images are used as the training set, and 2106 images used as the test set. In the training set, the primary vehicle was spawned at random locations in Town03, with the object to be detected at distances of -17.0m to 17.0m for parameter a and 0.1m to 3.0m for parameter b, as illustrated in Figure 9. The test set comprises of images generated according to 24 different cases of varying a, b distances set for the object to be detected, as shown in Table I. The position of the primary vehicle is also fixed at the same T-junction for all images. The images were used to train and test our custom deep learning based object detection model, which is based on FCOS object detection model [6].

a / b	0m	1m	2m	3m	10m	16.7m
0.1m	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
0.5m	Case 7	Case 8	Case 9	Case 10	Case 11	Case 12
1.5m	Case 13	Case 14	Case 15	Case 16	Case 17	Case 18
3.0m	Case 19	Case 20	Case 21	Case 22	Case 23	Case 24

TABLE I: The different cases for the test set images.

The machine used during training and testing is a server fitted with 4 Nvidia GeForce GTX Titan X graphic cards, along with an Intel(R) Xeon(R) E5-2640 2.40GHz CPU. The main goal of the evaluation is to see how accurate the sensing function can detect objects using a wide-angle sensor according to the Euro NCAP scenario as illustrated in Figure 9. Before training the model, the generated dataset was processed first. This processing was done in a few stages. First, each image in the dataset was processed using image transformation software. The software transforms each image in the dataset to obtain a flat panoramic image by applying an equirectangular projection. The transformed image is also resized to a resolution of 1280x2048 pixels as shown in Figure 13. Next,

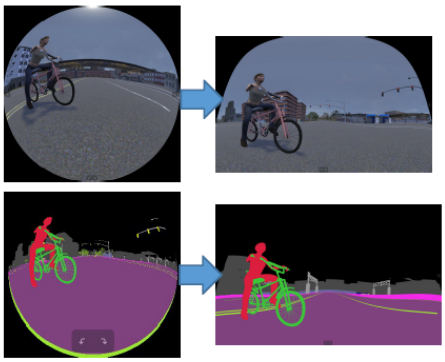


Fig. 13: Transforming the fisheye images.

the object to be detected, whether a pedestrian or a cyclist, has its bounding box extracted with a Python script based on openCV library [16]. The annotations are encoded in You Only Look Once(YOLO) [17] label format, so that any manual correction can later be performed using the open-source tool, Labelimg [18]. This tool only accepts PASCAL Visual Objects Challenge(VOC) [19] or YOLO annotation formats. Rain weather effects are also added to a subset of the images in the training set by blending them with rain using openCV. Finally, the dataset is transformed into Common Object in Context (COCO) format, a popular benchmark to use for testing new object detection models [20]. The transformed dataset is now suitable for use in the training and testing of the model. The model is trained with the parameters shown in Table II.

Scenario	Camera Settings	Objects	Position	No. of Images
Clear Noon	Pitch 35°, front of car	Pedestrian/Cylist	Random	1997
Clear Sunset	Pitch 35°, front of car	Pedestrian/Cylist	Random	1998
Cloudy Noon	Pitch 35°, front of car	Pedestrian/Cylist	Random	2009
Hard-Rainy Noon	Pitch 35°, front of car	Pedestrian/Cylist	Random	2013

TABLE II: Training Parameters

The testing set consists of roughly equal amounts of test images in different weather conditions. The images are generated with a virtual fisheye lens sensor that mimics the output of a real-world camera system fitted with a Sigma f3.5, 8mm, 180-degrees fisheye lens. The images contain either a pedestrian or cyclist at various distances as illustrated in Figure 9 and Table I. The following tables shows the mAP and AP50 evaluation metrics for the testing set.

Scenario	Number of images	mAP	AP50
Clear noon	528	85.7	98.5
Clear sunset	528	90.1	99.4
Cloudy noon	522	90.1	99.3
Hard rain noon	528	82.4	97.2

TABLE III: Overall evaluation results

b \ a	0m	1m	2m	3m	10m	16.7m
0.1m	91.8	96.6	95.2	96.0	96.0	92.0
0.5m	93.2	97.0	95.4	95.9	95.6	91.1
1.5m	91.1	96.7	95.4	96.6	95.8	96.3
3.0m	94.7	95.9	95.6	96.7	95.7	89.3

TABLE IV: AP50 scores for pedestrian

b \ a	0m	1m	2m	3m	10m	16.7m
0.1m	93.0	95.3	95.4	96.1	95.4	96.8
0.5m	95.0	95.3	95.4	96.9	95.1	97.0
1.5m	94.0	96.1	95.4	95.8	96.3	96.9
3.0m	91.1	95.3	97.0	96.4	96.3	96.3

TABLE V: AP50 scores for cyclist

From Table III, the highest mAP score was 90.1% for clear sunset and cloudy noon while the lowest score was 82.4% for hard rain noon. On the other hand, AP50 scores ranges from a high of 99.4% for clear sunset to 97.2% for hard rain noon. The lower score attained by hard rain is expected as rain has an effect on object visibility. From Table IV and Table V, it can be observed that the AP50 scores are consistently high across all cases. The lowest score for pedestrian is 89.3% and 91.1% for cyclist, while the highest score for pedestrian is 96.7% and 97.0% for cyclist. From these results, we can say that the model is able to make correct detections of the objects. This further shows that using a fisheye lens on a sensor is a feasible solution to detect objects at far perpendicular distances out of sight of sensors without fisheye lenses. These results also attest to the usability of our generated images as training and testing data for object detection models. Hence, it motivates the next step of creating more scenarios in the future such as detection of blind spot of motorists.

IV. Conclusion

In this paper, we have proposed a scenarios simulator with a customized fisheye lens with a distortion factor that can generate large training and testing datasets that mimics the real environment in order to alleviate the costs and speed up the process of developing autonomous vehicles. With our proposed implemented system that is able to adapt to any manufactured model, it was able to generate images at a rate of 720 images per hour which is 42.86% faster than existing methods. Additionally, the overall results of evaluating the object detection deep learning machine learning model on fisheye images is promising as they show high mAP and AP50 score of 90.1% and 99.4% in clear sky respectively. In terms of future work, the simulator could be expanded to include a broader range of scenarios. The usage of pix2pix image generation algorithm [21] could also be explored. The algorithm can be applied to our generated semantic segmentation images, creating realistic street images without relying on the RGB images. The resulting images may have the potential to be more realistic and closer to ground truth.

References

[1] S. Kumar and E. Goel, "Changing the world of autonomous vehicles using cloud and big data," in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pp. 368–376, 2018.

[2] S. W. Chen, C. K. Seow, S. Y. Tan, and P. B. De Silva, "Measurements and characterization of twisted radio wave multipath for indoor wireless communication and security system," in *2019 Photonics & Electromagnetics Research Symposium-Fall (PIERS-Fall)*, pp. 54–62, Dec 2019.

[3] R. Zhang, C. K. Seow, K. Wen, and H. Zhang, "Spoofing attack of drone," in *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, pp. 1239–1246, Dec 2018.

[4] H. Zhang, S. Y. Tan, and C. K. Seow, "TOA-based indoor localization and tracking with inaccurate floor map via MRMSC-PHD filter," *IEEE Sensors Journal*, vol. 19, pp. 9869–9882, Nov 2019.

[5] Y. J. J. Teoh and C. K. Seow, "RF and network signature-based machine learning on detection of wireless controlled drone," in *2019 Photonics & Electromagnetics Research Symposium-Spring (PIERS-Spring)*, pp. 408–417, June 2019.

[6] Z. Tian, C. Shen, H. Chen, and T. He, "Fcos: Fully convolutional one-stage object detection," in *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[7] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning* (S. Levine, V. Vanhoucke, and K. Goldberg, eds.), vol. 78 of *Proceedings of Machine Learning Research*, pp. 1–16, PMLR, 13–15 Nov 2017.

[8] S. Hossain, A. Fayjie, O. Doukhi, and D. Lee, *CAIAS Simulator: Self-driving Vehicle Simulator for AI Research*, pp. 187–195. 01 2019.

[9] S. Shah, A. Kapoor, D. Dey, and C. Lovett, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," *Field and Service Robotics*, pp. 621–635, November 2017.

[10] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *ECCV 2016*, Oct 2016.

[11] A. Gaidon, Q. Wang, Y. Cabon, and E. Vig, "Virtualworlds as proxy for multi-object tracking analysis," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4340–4349, 2016.

[12] D. Ott and T. Davis, "Simulating a virtual fisheye lens for the production of full-dome animations," pp. 294–299, 01 2007.

[13] P. Bourke, "idome: Immersive gaming with the unity3d game engine," in *CGAT09 Computer Games, Multimedia and Allied Technology 09 Proceedings* (E. Prakash, ed.), vol. 1, pp. 265–272, Research Publishing Services, 2009.

[14] P. Bourke, "Fisheye lens correction." http://paulbourke.net/dome/fisheyecorrect/, November 2016.

[15] P. Bourke, "Computer generated angular fisheye projections." http://paulbourke.net/dome/fisheye/, May 2001.

[16] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[17] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.

[18] Tzutalin, "Labelimg." https://github.com/tzutalin/labelImg, 2015.

[19] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, pp. 303–308, September 2009. Printed version publication date: June 2010.

[20] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and L. Zitnick, "Microsoft coco: Common objects in context," in *ECCV*, European Conference on Computer Vision, September 2014.

[21] P. Isola, J. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5967–5976, 2017.